GA4 BigQuery Pitfalls

14 traps teams hit when working with GA4 BigQuery exports — and how to avoid them.

Timo Dechau

April 2026

Introduction

So you've enabled the GA4 BigQuery export. Nice. You're already ahead of most teams — you've got raw event data sitting in BigQuery, ready to be queried.

Now here's the thing: the gap between "export is enabled" and "we have a reliable data model we actually trust" is where teams lose weeks. Sometimes months. I've seen it happen over and over again across client engagements.

The GA4 BigQuery export is genuinely powerful. But it's also full of traps. Some are obvious once you hit them — like that first terrifying BigQuery bill. Others are subtle — the kind of thing where your numbers look plausible but are quietly wrong, and you don't find out until someone asks a question you can't answer.

This guide covers 14 pitfalls I've seen repeatedly. Not theoretical risks — actual things that actual teams run into. Each one follows the same structure:

- **The Trap** — what people do wrong and why it seems reasonable
- **Why It Happens** — what about GA4 or BigQuery makes this non-obvious
- **The Fix** — a concrete solution, usually with SQL you can run
- **How to Check If You Have This Problem** — a diagnostic query or quick check you can run against your own data right now

That last part is the important one. I didn't want this to be something you read, nod along to, and forget. Every pitfall gives you something to actually do — run a query, check a setting, have a conversation with your team. That's what makes it useful.

The pitfalls are organized in the order you'll likely encounter them: infrastructure and cost surprises first, then schema and export fundamentals, session and identity challenges, a couple of mindset shifts that save enormous amounts of wasted effort, and finally the data modeling principles that tie it all together.

At the end, you'll find the Attribution Readiness Scorecard — five questions to honestly assess where your organization stands on the road from "raw GA4 data in BigQuery" to "attribution we actually trust." It's a good conversation starter with your team.

Let's get into it.

> ⓘNote
>
> **A note on currency:** GA4's BigQuery export schema evolves. Everything in this guide is current as of March 2026. If you're reading this later, the principles hold — but double-check the specifics against Google's documentation.

> 🚀Tip
>
> **A note on export limits:** For standard (non-360) GA4 properties, the streaming export is limited to 1 million events per day. The daily export limit has been effectively removed for most users, though Google's "best effort" policies still apply — very high-volume sites may encounter overflow tables. If you're on a 360 property, these limits are significantly higher. Either way, check your event volume against these thresholds early, before you build pipelines that assume every event arrives.

Pitfall #1: Sandbox Mode Silently Expires Your Data

The Trap

Section titled “The Trap”

You set up the GA4 BigQuery export using BigQuery’s free sandbox tier. Everything works. Data flows in daily. You feel good about it — you’re collecting raw event data without spending a cent. Then one day, months later, you go to run a year-over-year comparison and discover your oldest data is just… gone. No warning, no error message, no email from Google. It simply expired.

Why It Happens

Section titled “Why It Happens”

BigQuery’s sandbox mode enforces a 60-day maximum table expiration. When your GA4 export creates daily tables like `events_20250115`, sandbox automatically stamps each one with a 60-day time-to-live. That’s the deal — free tier, limited retention.

Here’s where it gets nasty. When you eventually enable billing (and you will), BigQuery does *not* go back and remove those expiration settings from your existing tables. The dataset-level default expiration stays set. Every table that was created under sandbox rules keeps its original expiration timestamp. Your new tables might be fine, but your historical data is still on a countdown timer.

People discover this in one of two ways: they notice a gap in their historical data, or — worse — they don’t notice at all and just assume their dataset starts when they think it does.

The source of this gotcha is well-documented on ga4bigquery.com, and I’ve personally seen it catch multiple teams who assumed “enabling billing” meant “problem solved.”

The Fix

Section titled “The Fix”

When you move from sandbox to a billing-enabled project, you need to explicitly clean up expiration settings at two levels:

1. Remove the dataset-level default expiration:

```
ALTER SCHEMA `your_project.analytics_123456789`
SET OPTIONS (
  default_table_expiration_days = NULL
);
```

2. Remove expiration from all existing event tables:

```
DECLARE tables ARRAY<STRING>;
DECLARE i INT64 DEFAULT 0;

SET tables = (
  SELECT ARRAY_AGG(table_name)
  FROM
    `your_project.analytics_123456789`
      .INFORMATION_SCHEMA.TABLES
  WHERE table_name LIKE 'events_%'
);

WHILE i < ARRAY_LENGTH(tables) DO
  EXECUTE IMMEDIATE FORMAT(
    'ALTER TABLE `your_project.analytics_123456789.%s`'
    || ' SET OPTIONS (expiration_timestamp = NULL)',
    tables[OFFSET(i)]
  );
  SET i = i + 1;
END WHILE;
```

Yes, you need to do both. The dataset-level change only affects *future* tables. Every existing table retains whatever expiration was set when it was created.

How to Check If You Have This Problem

Section titled “How to Check If You Have This Problem”

Run this against your GA4 dataset right now:

```sql
SELECT
  table_name,
  option_name,
  option_value
FROM
  `your_project.analytics_123456789`
    .INFORMATION_SCHEMA.TABLE_OPTIONS
WHERE
  option_name = 'expiration_timestamp'
  AND table_name LIKE 'events_%'
ORDER BY table_name
LIMIT 20;
```

If this returns rows, those tables have expiration dates set. If the timestamps are in the past, those tables are already gone. If they're in the future, you still have time — but act now.

Also check the dataset-level default:

```sql
SELECT
  option_name,
  option_value
FROM
  `your_project.analytics_123456789`
    .INFORMATION_SCHEMA.SCHEMATA_OPTIONS
WHERE
  option_name = 'default_table_expiration_days';
```

If this returns a value, every new table created in the dataset will inherit that expiration. Remove it.

Look, this one stings because it's entirely preventable. Five minutes of checking these settings after enabling billing can save you from losing months of irreplaceable historical data.

Pitfall #2: LIMIT Doesn't Limit Your Bill

The Trap

Section titled “The Trap”

You’re new to BigQuery — or maybe just new to GA4’s export — and you want to explore the data. So you do the sensible thing: `SELECT * FROM events_* LIMIT 10`. Just ten rows, right? How expensive could that be? Very. That query just scanned your entire dataset. Every column, every table, every day of data you’ve ever collected. The `LIMIT 10` only limits the *output*. BigQuery already read everything to get there.

Why It Happens

Section titled “Why It Happens”

BigQuery is a columnar, full-scan engine. When you run a query, it reads all the data that matches your `FROM` clause before applying `LIMIT`. That’s just how the architecture works — there’s no early-exit optimization that says “I found 10 rows, I can stop now.”

Now here’s where it gets interesting with GA4 specifically. Your GA4 export doesn’t use a single partitioned table. It creates date-sharded tables: `events_20250101`, `events_20250102`, and so on. When you query `events_*` with a wildcard, BigQuery unions *all* of those tables together. Without a `_TABLE_SUFFIX` filter to constrain which date-shards get included, you’re scanning everything. Every single day.

And this compounds. On day one, your dataset is small. By month six, it might be hundreds of gigabytes. By year two, a terabyte or more. The exact same `SELECT *` query gets more expensive every single day, even if you never change it. Scheduled queries and dashboards that seemed cheap in January can quietly become budget problems by October.

The Fix

Section titled “The Fix”

Three habits that will save you real money:

1. Always filter by `_TABLE_SUFFIX` for date ranges:

```
SELECT
  event_name,
  event_timestamp,
  user_pseudo_id
FROM
  `your_project.analytics_123456789.events_*`
WHERE
  _TABLE_SUFFIX BETWEEN '20260301' AND '20260325'
```

This tells BigQuery which date-sharded tables to actually read. Everything outside the range is skipped entirely — zero bytes scanned.

2. Use `TABLESAMPLE` for exploration:

```
SELECT *
FROM
  `your_project.analytics_123456789.events_20260325`
  TABLESAMPLE SYSTEM (1 PERCENT)
```

This physically reads only ~1% of the table's storage blocks. It's perfect for "I just want to see what the data looks like" moments. Not statistically rigorous, but that's not the point — you're exploring, not analyzing.

3. Always check dry-run bytes before executing:

In the BigQuery console, look at the green validator message in the top-right corner before you hit Run. It tells you exactly how many bytes the query will scan. Get in the habit of glancing at it. If a "quick exploration" query says it'll scan 800 GB, that's your cue to add a date filter.

You can also dry-run from the command line:

```
bq query --dry_run --use_legacy_sql=false \
  'SELECT event_name
   FROM `your_project.analytics_123456789.events_*`
   WHERE _TABLE_SUFFIX = "20260325"'
```

How to Check If You Have This Problem

Section titled "How to Check If You Have This Problem"

Run the same simple query with and without a date filter, using dry-run mode, and compare the bytes:

```
-- Without date filter (check dry-run bytes):
SELECT event_name
FROM `your_project.analytics_123456789.events_*`
LIMIT 10;

-- With date filter (check dry-run bytes):
SELECT event_name
FROM `your_project.analytics_123456789.events_*`
WHERE _TABLE_SUFFIX = '20260325'
LIMIT 10;
```

Don't actually run the first one — just look at the dry-run estimate. If the difference is 100x or more, you now understand why every query needs a date filter. That gap is money.

Also check your query history for past damage:

```
SELECT
  user_email,
  query,
  total_bytes_processed,
  ROUND(
    total_bytes_processed / POW(1024, 3), 2
  ) AS gb_processed,
  creation_time
FROM
  `region-us`.INFORMATION_SCHEMA.JOBS_BY_PROJECT
WHERE
  creation_time > TIMESTAMP_SUB(
    CURRENT_TIMESTAMP(), INTERVAL 7 DAY
  )
  AND total_bytes_processed > 10 * POW(1024, 3)
ORDER BY total_bytes_processed DESC
LIMIT 20;
```

This surfaces any queries from the last week that scanned more than 10 GB. You might be surprised what you find.

Pitfall #3: No Quota = No Safety Net

The Trap

Section titled “The Trap”

You’ve enabled billing on your BigQuery project. Maybe you’ve even set up a budget alert or two. But you haven’t touched the query quota settings, because honestly, you didn’t know they existed. Then someone on your team runs a bad query — a full scan across two years of GA4 data, joined to itself, no date filter. Or a scheduled query breaks and starts retrying in a loop. Google’s default daily query limit for BigQuery is 200 TB. At on-demand pricing, that’s roughly $1,250 in a single day. And that’s per *project*, not per organization — so multiply it by however many projects have billing enabled.

Why It Happens

Section titled “Why It Happens”

Google sets generous defaults because they’re optimizing for “it just works” — not for “protect the customer’s wallet.” 200 TB per day is a limit most people will never hit through normal usage. But “normal usage” and “someone made a mistake” are very different things. A recursive CTE that blows up, a BI tool that generates cartesian joins behind the scenes, an intern exploring the data with `SELECT *` — these things happen. And without a quota, there’s nothing between that mistake and your credit card.

The other issue is that budget alerts in GCP are *notifications*, not *enforcement*. A budget alert at $100/month will send you an email when you cross the threshold. It will not stop the query. It will not prevent the next one. By the time you read that email, the damage may already be done.

The Fix

Section titled “The Fix”

Set two types of quotas immediately:

1. Project-level daily query quota:

Go to GCP Console > IAM & Admin > Quotas (or search for “BigQuery API” quotas). Find “Query usage per day” and set it to something reasonable. For most GA4 analytics

workloads, 1 TB/day is a sensible starting point. That's about $6.25/day at on-demand pricing. You can always increase it later — but you can't un-scan terabytes.

2. Per-user daily query quota:

In the same quota settings, find "Query usage per day per user." Set this lower than the project quota — maybe 500 GB or even 100 GB. This way, one person's mistake can't consume the entire project's budget.

3. Set up budget alerts at multiple thresholds:

Go to Billing > Budgets & Alerts and create a budget for the project:

- Alert at 50% of your monthly budget
- Alert at 80%
- Alert at 100%
- Alert at 200% (yes, you can exceed your budget — alerts don't stop spending)

Here's the thing — these alerts are necessary but not sufficient. They tell you there's a fire. The quotas are what actually prevent the fire from burning the house down.

4. Consider slot-based pricing for predictability:

If your team queries GA4 data heavily, BigQuery Editions (with committed slots) give you a flat-rate pricing model. You buy a fixed amount of compute capacity. Queries might queue or run slower if you exceed your slots, but your bill stays predictable. For teams running lots of scheduled queries and dashboards against GA4 data, this often ends up cheaper *and* safer than on-demand pricing.

How to Check If You Have This Problem

Section titled "How to Check If You Have This Problem"

Go to GCP Console and check your current settings:

1. **Check query quotas:** Navigate to IAM & Admin > Quotas > filter for "BigQuery API" > look for "Query usage per day." If it says 200 TB (or unlimited), you're running without a safety net.
2. **Check budget alerts:** Navigate to Billing > Budgets & Alerts. If you see nothing there, you have no alerts configured at all.
3. **Check what you've actually been spending:**

```sql
SELECT
  DATE(creation_time) AS query_date,
  user_email,
  COUNT(*) AS query_count,
  ROUND(
    SUM(total_bytes_processed) / POW(1024, 4), 3
  ) AS tb_scanned,
  ROUND(
    SUM(total_bytes_processed)
      / POW(1024, 4) * 6.25,
    2
  ) AS estimated_cost_usd
FROM
  `region-us`.INFORMATION_SCHEMA.JOBS_BY_PROJECT
WHERE
  creation_time > TIMESTAMP_SUB(
    CURRENT_TIMESTAMP(), INTERVAL 30 DAY
  )
  AND job_type = 'QUERY'
  AND state = 'DONE'
GROUP BY query_date, user_email
ORDER BY tb_scanned DESC
LIMIT 30;
```

This shows your actual query volume by user over the last 30 days. Look at the `estimated_cost_usd` column. If any single day or user surprises you, that's exactly why you need quotas.

I'll admit, setting quotas feels like busywork when you're excited to start analyzing data. But I've seen a single afternoon of careless queries cost more than a team's entire monthly analytics budget. Five minutes in the GCP Console now can save you a very uncomfortable conversation with finance later.

Pitfall #4: Treating events_* Like a Flat Table

The Trap

Section titled “The Trap”

You open the GA4 export in BigQuery, write `SELECT * FROM your_project.analytics_123456789.events_*`, and immediately something feels off. Some columns return `RECORD` as their type. Your row counts don’t match what you expected. Or you try to filter on a specific event parameter and BigQuery throws an error about accessing fields on a repeated struct. So you start guessing at syntax, copy-pasting UNNEST patterns from Stack Overflow, and suddenly your 50,000 events became 300,000 rows and you have no idea why.

Why It Happens

Section titled “Why It Happens”

If you’ve spent your career working with flat, relational tables — which most analysts have — GA4’s export schema is a genuine shock. Three critical columns are not scalar values but nested, repeated RECORD fields:

- `event_params` — a repeated STRUCT containing every parameter attached to an event (page_location, ga_session_id, engagement_time, and dozens more)
- `user_properties` — a repeated STRUCT of user-scoped properties set via the SDK
- `items` — a repeated STRUCT for e-commerce item data

“Repeated” means each event row can contain an array of these records. A single `page_view` event might carry 15 different event parameters, each stored as a separate element inside the `event_params` array. That’s not 15 rows in the table — it’s 15 elements nested inside one row.

The confusion deepens because `event_params` doesn’t use named columns. Every parameter is stored as a generic key-value pair with `key` (string) and `value` (another STRUCT with `string_value`, `int_value`, `float_value`, `double_value`). So there’s no `event_params.page_location` column you can just reference. You have to dig into the array, find the element where `key = 'page_location'`, and then pull out the correct value field.

This design is flexible — GA4 can add any parameter without changing the schema — but it's hostile to anyone who just wants to write a WHERE clause.

The Fix

Section titled "The Fix"

There are two patterns you'll use constantly. Know when to reach for each one.

Pattern 1: Extract a single parameter (no row multiplication)

Use a subquery to pluck one value out of the array. This keeps your row count intact — one row in, one row out.

```
SELECT
  event_timestamp,
  event_name,
  (SELECT value.string_value
   FROM UNNEST(event_params)
   WHERE key = 'page_location')
    AS page_location,
  (SELECT value.int_value
   FROM UNNEST(event_params)
   WHERE key = 'ga_session_id')
    AS ga_session_id
FROM
  `your_project.analytics_123456789.events_*`
WHERE
  _TABLE_SUFFIX BETWEEN '20260301' AND '20260307'
```

Each `(SELECT ... FROM UNNEST(...))` subquery runs independently against the nested array for that row. It returns exactly one value (or NULL if the key doesn't exist). Your output has the same number of rows as your input. This is the pattern you want 90% of the time.

Pattern 2: Explode items for e-commerce analysis (intentional row multiplication)

When you actually need one row per item — say, for product-level revenue analysis — use `CROSS JOIN UNNEST`:

```sql
SELECT
  event_timestamp,
  event_name,
  items.item_name,
  items.item_revenue,
  items.quantity
FROM
  `your_project.analytics_123456789.events_*`,
  UNNEST(items) AS items
WHERE
  _TABLE_SUFFIX BETWEEN '20260301' AND '20260307'
  AND event_name = 'purchase'
```

Here's the thing: `CROSS JOIN UNNEST` (which the comma syntax is shorthand for) multiplies your rows. A purchase event with 3 items becomes 3 rows. That's exactly what you want for item-level analysis — but if you do this accidentally on `event_params`, your row count explodes and your aggregations break.

The rule of thumb: use the subquery pattern for `event_params` and `user_properties`. Use `CROSS JOIN UNNEST` for `items` only when you specifically need item-level granularity.

How to Check If You Have This Problem

Section titled "How to Check If You Have This Problem"

Run this query to see the difference side by side. It compares the base event count with what happens when you accidentally unnest `event_params`:

```sql
WITH base AS (
  SELECT COUNT(*) AS event_count
  FROM
    `your_project.analytics_123456789.events_*`
  WHERE
    _TABLE_SUFFIX BETWEEN '20260301' AND '20260301'
),
unnested AS (
  SELECT COUNT(*) AS exploded_count
  FROM
    `your_project.analytics_123456789.events_*`,
    UNNEST(event_params) AS params
  WHERE
    _TABLE_SUFFIX BETWEEN '20260301' AND '20260301'
)
SELECT
  base.event_count,
  unnested.exploded_count,
  ROUND(
    unnested.exploded_count / base.event_count, 1
  ) AS multiplication_factor
FROM base, unnested
```

You'll typically see a multiplication factor between 10x and 20x. That's 10-20 event parameters per event, each becoming its own row. If any of your production queries use `CROSS JOIN UNNEST(event_params)` without a very good reason, you've found your problem.

Pitfall #5: Misunderstanding the Intraday Table

The Trap

Section titled “The Trap”

You’re building a dashboard that needs “today’s” data. You query `events_*` with today’s date in the `_TABLE_SUFFIX` filter and get… nothing. Or you get data, but it’s from yesterday. So you dig around, discover `events_intraday_*`, and start querying that instead. Problem solved, right? Until tomorrow, when someone notices your numbers for yesterday are either missing events or double-counting them. You’ve walked into one of GA4’s quieter traps.

Why It Happens

Section titled “Why It Happens”

GA4’s BigQuery export actually produces two sets of tables, and they have a lifecycle most people never read about:

1. **`events_intraday_YYYYMMDD`** — Created during the day. Contains events as they stream in, refreshed multiple times per hour. This is your “live-ish” data. It’s fresh but incomplete.
2. **`events_YYYYMMDD`** — Created once Google finishes processing the full day’s data. This is the final, complete table for that date.

Here’s the critical part: when `events_20260325` is finalized, `events_intraday_20260325` gets deleted. The intraday table is temporary. It exists only while the day is still being processed.

The timing of this handoff is where things get messy. Google doesn’t guarantee when the daily table will appear. It’s usually within 24-72 hours after the day ends, but I’ve seen it take longer. During that gap, the intraday table is your only source for that date. After the handoff, it’s gone and the daily table is your only source.

Two common mistakes:

- **Querying only `events_*`** and wondering why “today” has no data (because the daily table for today doesn’t exist yet — the intraday table does)

- **Querying both `events_*` and `events_intraday_*`** with a UNION ALL and accidentally double-counting the overlap period where both exist for the same date

The Fix

Section titled “The Fix”

The cleanest pattern is a query that checks both tables but avoids overlap. The logic: for any given date, use the daily table if it exists; fall back to intraday only if the daily table doesn’t exist yet.

```
SELECT * FROM (
  -- Final daily tables
  SELECT
    event_timestamp,
    event_name,
    user_pseudo_id,
    event_params,
    'daily' AS source_table
  FROM
    `your_project.analytics_123456789.events_*`
  WHERE
    _TABLE_SUFFIX BETWEEN '20260320' AND '20260325'

  UNION ALL

  -- Intraday for dates not yet finalized
  SELECT
    event_timestamp,
    event_name,
    user_pseudo_id,
    event_params,
    'intraday' AS source_table
  FROM
    `your_project.analytics_123456789.events_intraday_*`
  WHERE
    _TABLE_SUFFIX BETWEEN '20260320' AND '20260325'
    AND _TABLE_SUFFIX NOT IN (
      SELECT DISTINCT _TABLE_SUFFIX
      FROM
        `your_project.analytics_123456789.events_*`
      WHERE
        _TABLE_SUFFIX BETWEEN '20260320' AND '20260325'
    )
)
```

The subquery in the `NOT IN` clause ensures you only pull intraday data for dates where no finalized daily table exists. Once the daily table lands, the intraday rows for that date are excluded automatically.

For scheduled pipelines, the more robust approach is to check `INFORMATION_SCHEMA.TABLES` for table existence before deciding which to query. But for ad-hoc analysis, the pattern above is reliable and readable.

One more thing: if you're building materialized models (say, in dbt or Dataform), design your incremental logic around this handoff. Process intraday data provisionally, then reprocess once the daily table drops. The daily table is authoritative — it may contain events that the intraday table missed, and it applies final processing that intraday doesn't.

> Tip
>
> **Late-arriving events matter.** Google can append events to `events_*` tables for up to 72 hours after the day ends. A query you run today for "yesterday" might return slightly different results if you run it again three days later. For any analysis that requires precision — attribution, conversion counts, financial reporting — wait at least 72 hours before treating a day's data as final. Better yet, build your pipeline to reprocess the last 3 days on every run.

> Note
>
> **Watch for missing parameters on `session_start` events.** The very first event in a session — typically `session_start` or `first_visit` — sometimes arrives without `page_location` or `page_referrer` parameters if the GA4 SDK hasn't fully initialized when the event fires. If you're building session-level attribution by pulling the source from the first event, you may get NULL values for a small percentage of sessions. Check for this explicitly and consider falling back to the next event in the session when the first one is missing these fields.

How to Check If You Have This Problem

Section titled "How to Check If You Have This Problem"

Run this to compare row counts between the intraday and daily table for the most recent date where both exist. If both return rows for the same date, you need the deduplication pattern:

```sql
SELECT
  'daily' AS table_type,
  _TABLE_SUFFIX AS date_shard,
  COUNT(*) AS row_count
FROM
  `your_project.analytics_123456789.events_*`
WHERE
  _TABLE_SUFFIX >= FORMAT_DATE(
    '%Y%m%d', DATE_SUB(CURRENT_DATE(), INTERVAL 3 DAY)
  )
GROUP BY _TABLE_SUFFIX

UNION ALL

SELECT
  'intraday' AS table_type,
  _TABLE_SUFFIX AS date_shard,
  COUNT(*) AS row_count
FROM
  `your_project.analytics_123456789.events_intraday_*`
WHERE
  _TABLE_SUFFIX >= FORMAT_DATE(
    '%Y%m%d', DATE_SUB(CURRENT_DATE(), INTERVAL 3 DAY)
  )
GROUP BY _TABLE_SUFFIX

ORDER BY date_shard, table_type
```

Look at the output. For recent dates, you should see either a daily row or an intraday row — not both. If you see both for the same `date_shard`, that's the overlap window. Any query spanning that period without the deduplication logic above is double-counting.

Pitfall #6: Trusting event_date Instead of event_timestamp

The Trap

Section titled “The Trap”

There’s a friendly-looking column called `event_date` right there in the schema. It’s a string, formatted `YYYYMMDD`, easy to work with. You use it to filter, group, and aggregate your events by day. Your reports look fine. Then one day, you’re analyzing conversions by hour and you notice something: events that clearly happened at 11 PM on March 24th are showing up under March 25th. Your daily totals have been subtly wrong this entire time.

Why It Happens

Section titled “Why It Happens”

Two things are working against you here, and they compound each other.

First: `event_date` is the date the event was *processed and assigned to a daily table shard*, not necessarily the date the event *occurred*. For most events, they’re the same. But for late-arriving hits — events from offline devices that sync later, or events buffered on mobile before being sent — the event can land in a table shard that doesn’t match when it actually happened. GA4 also applies its own processing logic that can shift which date an event is attributed to.

Second, and this is the bigger one: everything in the GA4 BigQuery export is in UTC. The `event_timestamp` field is microseconds since Unix epoch, in UTC. The `event_date` column is also derived in UTC. If your business operates in, say, `Europe/Berlin` (UTC+1 or UTC+2 depending on DST), then an event that happened at 11:30 PM Berlin time on March 24th has a UTC timestamp of March 25th 10:30 PM — wait, no. Let me be precise. An event at 11:30 PM CET (UTC+1) is 10:30 PM UTC on the same day. But an event at 00:30 AM CET on March 25th is 11:30 PM UTC on March 24th.

Here’s the thing: depending on your timezone offset direction, events get shifted forward or backward relative to your business day. The further you are from UTC, the bigger the mismatch. For teams in US Pacific time (UTC-7 or UTC-8), the shift is massive — nearly a third of your business day falls on the “wrong” UTC date.

And `event_date` gives you no way to fix this. It's a pre-computed string. You can't adjust it for timezone. You're stuck with whatever date Google assigned.

The Fix

Section titled "The Fix"

Always use `event_timestamp` for any time-based analysis. Convert it from microseconds to a proper TIMESTAMP, then apply your business timezone explicitly.

```
SELECT
  event_name,
  TIMESTAMP_MICROS(event_timestamp) AS event_ts_utc,
  DATETIME(
    TIMESTAMP_MICROS(event_timestamp),
    'Europe/Berlin'
  ) AS event_datetime_berlin,
  DATE(
    TIMESTAMP_MICROS(event_timestamp),
    'Europe/Berlin'
  ) AS event_date_berlin,
  event_date AS event_date_utc_original
FROM
  `your_project.analytics_123456789.events_*`
WHERE
  _TABLE_SUFFIX BETWEEN '20260301' AND '20260307'
LIMIT 100
```

A few rules to internalize:

1. **Use `event_timestamp` for grouping by date, hour, or any time window.** Convert to your business timezone first.
2. **Keep `_TABLE_SUFFIX` for partition pruning only.** It's a cost optimization tool, not an analytical date field. When you filter by `_TABLE_SUFFIX`, widen the range by a day on each side to catch edge cases, then apply your real date filter on the converted timestamp.
3. **Be explicit about timezone in every query.** Never assume UTC is "close enough." Write `DATE(TIMESTAMP_MICROS(event_timestamp), 'America/New_York')` instead of `event_date`. Every time.
4. **Store the timezone conversion in your models.** If you're building intermediate tables, compute the business-timezone date once and include it as a column. Don't make every downstream query re-derive it.

How to Check If You Have This Problem

Section titled “How to Check If You Have This Problem”

This query finds events where the UTC-derived `event_date` and the business-timezone date disagree. Replace `'Europe/Berlin'` with your own timezone:

```
SELECT
  event_date AS table_date_utc,
  DATE(
    TIMESTAMP_MICROS(event_timestamp),
    'Europe/Berlin'
  ) AS actual_date_berlin,
  COUNT(*) AS event_count
FROM
  `your_project.analytics_123456789.events_*`
WHERE
  _TABLE_SUFFIX BETWEEN '20260301' AND '20260307'
GROUP BY 1, 2
HAVING table_date_utc != CAST(actual_date_berlin AS STRING)
ORDER BY event_count DESC
```

If this returns rows, those are events that would land on a different date in your business reporting than where `event_date` puts them. The `event_count` tells you how many events are affected. For European timezones, expect a small but consistent percentage. For US timezones, the numbers can be significant — especially for events near midnight UTC, which is late afternoon or early evening in the US.

Any report or model that groups by `event_date` instead of a timezone-converted timestamp is carrying this error silently.

Pitfall #7: Building Audience Segments from Events When GA4 Already Exports Them

The Trap

Section titled “The Trap”

Your marketing team wants to know which users are likely to churn. Or which users belong to the “high-value purchasers” audience you painstakingly built in the GA4 interface. So you do what any self-respecting analyst would do: you write a massive SQL query against `events_*`, reconstruct the user journey, apply your own logic to define the segment, and deliver the results. It takes two days and the numbers don’t quite match what the GA4 UI shows.

Meanwhile, GA4 has been quietly exporting exactly this data into tables you never looked at.

Why It Happens

Section titled “Why It Happens”

Most teams set up the GA4 BigQuery link, see the `events_*` tables appear, and stop there. That’s the export everyone talks about. Blog posts, tutorials, conference talks — it’s all about the event data.

But GA4 can also export two additional table types:

- **`pseudonymous_users_*`** — One row per `user_pseudo_id` (the cookie-based device identifier). Contains user properties, audience memberships, and Google’s predictive metrics.
- **`users_*`** — One row per `user_id` (your authenticated user identifier, if you set one). Same structure but scoped to known users.

These tables include fields that are genuinely difficult or impossible to reconstruct from raw events:

- **Audience memberships** — Every GA4 audience the user currently belongs to, with the timestamp they joined. This is the same audience logic the GA4 UI uses, including Google’s internal processing.

- **Predictive metrics** — Purchase probability, churn probability, and predicted revenue. These are machine learning scores Google computes on their side. You can't replicate them from the event stream because the model is proprietary.
- **User properties** — The full set of user-scoped properties, already resolved to their most recent values (no need to deduplicate from the event stream).
- **Lifetime metrics** — LTV revenue, session count, engagement metrics, all pre-computed.

Look, I'll admit: for a long time I didn't pay much attention to these tables either. They were introduced later, documentation was sparse, and the event tables were already there. But ignoring them means you're rebuilding wheels — and some of those wheels (predictive scores, audience membership timestamps) you literally cannot rebuild from events alone.

The Fix

Section titled "The Fix"

Enabling these exports takes about 30 seconds.

1. Go to **GA4 Admin** > **Product Links** > **BigQuery Links**
2. Click on your existing BigQuery link
3. Under **Configure data streams and events**, you'll see the export settings
4. Check the boxes for **"Include user data in daily export"** — this enables both `pseudonymous_users_*` and `users_*` tables
5. Make sure **"Daily"** frequency is selected (streaming export is also available but not needed for most use cases)
6. Save

Now here's where it gets interesting — and where the caveats live.

Caveat 1: No historical backfill. These tables start populating from the day you enable them. If you enable them on March 25th, your first `pseudonymous_users_20260325` table appears the next day. There's no retroactive export for users who were active before that date. Enable it now, even if you don't plan to use it immediately.

Caveat 2: User counts won't match event-derived counts. The `pseudonymous_users_*` table contains users who were active during the export window, based on Google's internal definition of "active." If you count distinct `user_pseudo_id` values from `events_*` for the same date range, you'll get a different

number. This is expected — the scoping logic is different. Don't try to reconcile them; use each table for its intended purpose.

Caveat 3: Audience membership reflects a point-in-time snapshot. The export captures which audiences a user belongs to at the time of export. It doesn't give you a full history of audience membership changes. If a user left and rejoined an audience, you'll only see the current membership.

Caveat 4: Predictive metrics require minimum thresholds. Google's purchase probability, churn probability, and revenue prediction models only activate if your property has enough data (roughly 1,000 purchasers and 1,000 churners in the past 28 days, plus other conditions). Smaller properties will see these fields as NULL.

How to Check If You Have This Problem

Section titled "How to Check If You Have This Problem"

First, check whether you even have these tables:

```
SELECT
  table_name,
  TIMESTAMP_MILLIS(creation_time) AS created,
  ROUND(size_bytes / POW(1024, 2), 1)
    AS size_mb,
  row_count
FROM
  `your_project.analytics_123456789`
    .__TABLES__
WHERE
  table_name LIKE 'pseudonymous_users_%'
  OR table_name LIKE 'users_%'
ORDER BY table_name DESC
LIMIT 10
```

If this returns no rows, the user export isn't enabled. Go enable it.

If it does return rows, take a look at what you've got:

```sql
SELECT
  user_pseudo_id,
  audiences,
  predictions,
  user_properties,
  user_ltv
FROM
  `your_project.analytics_123456789`
    .`pseudonymous_users_*`
WHERE
  _TABLE_SUFFIX = FORMAT_DATE(
    '%Y%m%d',
    DATE_SUB(CURRENT_DATE(), INTERVAL 2 DAY)
  )
LIMIT 20
```

Browse those results. If you see audience arrays populated with the segments you've built in the GA4 interface, and prediction scores with actual values instead of NULLs, you've got a data source that would have taken weeks to approximate from raw events. If predictions are all NULL, check whether your property meets Google's minimum thresholds — that's a volume issue, not a configuration one.

Either way, now you know what's there. And knowing is considerably better than spending two days writing SQL to approximate it.

Pitfall #8: user_pseudo_id Is Not a Stable User Identity

The Trap

Section titled “The Trap”

You start building your user-level analysis — attribution, lifetime value, retention — and you anchor it all on `user_pseudo_id`. It’s right there in every row of the export, it looks like a user identifier, and it feels like the obvious primary key for “a person.” So you count distinct `user_pseudo_id` values and call that your user count. You join tables on it. You build cohorts with it. And then one day someone asks why your BigQuery user count is 3x higher than what GA4 reports, or why a single customer appears to have five separate journeys that never connect.

Why It Happens

Section titled “Why It Happens”

The `user_pseudo_id` is a client ID — specifically, the value from the `_ga` cookie in a browser, or the app instance ID on mobile. That’s it. It identifies a browser on a device, not a person.

Here’s where the gap opens up. The same person using Chrome on their laptop, Safari on their phone, and the company iPad for weekend browsing? That’s three `user_pseudo_id` values. One person, three “users” in your data. Now multiply that across your entire user base.

It gets worse. Even on a single browser, the `user_pseudo_id` resets when someone clears their cookies, uses incognito mode, or when the browser itself decides to expire first-party cookies (Safari’s ITP does this after 7 days of no visit in some scenarios). Every reset mints a brand-new `user_pseudo_id`, and there’s nothing in the raw export that connects the old one to the new one.

People build entire attribution models on this identifier without realizing they’re actually measuring browser instances, not people. Your “new user” count is quietly inflated by returning visitors on fresh cookies.

The Fix

Section titled “The Fix”

Treat `user_pseudo_id` for what it actually is: a device-and-browser identifier. It’s useful — it’s the most granular identifier GA4 gives you out of the box — but it’s not a person.

For actual user identity, you need your own `user_id`. When someone logs in, GA4 lets you set this via `gtag('set', 'user_id', 'your-id')` or through the config. That value shows up in the export as `user_id`. It’s your system’s identifier, so it follows the person across devices and browsers.

The practical approach is to build an identity resolution layer:

```
-- Build a device-to-user mapping from login events
WITH identity_map AS (
  SELECT
    user_pseudo_id,
    user_id,
    MIN(event_timestamp) AS first_seen,
    MAX(event_timestamp) AS last_seen
  FROM
    `your_project.analytics_123456789.events_*`
  WHERE
    user_id IS NOT NULL
  GROUP BY
    user_pseudo_id, user_id
)

SELECT * FROM identity_map
ORDER BY user_id, first_seen;
```

This gives you a mapping table: which `user_pseudo_id` values belong to which actual `user_id`. Join this into your analyses and you’ll start seeing real people instead of cookie fragments.

For the portion of your traffic that never logs in, `user_pseudo_id` is the best you’ve got. Accept that limitation — just don’t confuse it with knowing who someone is.

How to Check If You Have This Problem

Section titled “How to Check If You Have This Problem”

Run this to see how much identity coverage you actually have:

```sql
SELECT
  COUNT(DISTINCT user_pseudo_id)
    AS total_pseudo_ids,
  COUNT(DISTINCT user_id)
    AS total_user_ids,
  COUNT(DISTINCT CASE
    WHEN user_id IS NOT NULL
    THEN user_pseudo_id
  END) AS pseudo_ids_with_user_id,
  ROUND(
    COUNT(DISTINCT CASE
      WHEN user_id IS NOT NULL
      THEN user_pseudo_id
    END)
    / COUNT(DISTINCT user_pseudo_id) * 100, 1
  ) AS identity_coverage_pct
FROM
  `your_project.analytics_123456789.events_*`
WHERE
  _TABLE_SUFFIX BETWEEN
    FORMAT_DATE('%Y%m%d', DATE_SUB(CURRENT_DATE(), INTERVAL 30 DAY))
    AND FORMAT_DATE('%Y%m%d', CURRENT_DATE());
```

Look at that `identity_coverage_pct` number. If it's below 20%, your user-level analysis is mostly fiction — you're counting cookies, not people. Even at 50%, you should be treating any "per-user" metric with serious caveats.

The gap between `total_pseudo_ids` and `total_user_ids` tells you how fragmented your identity picture really is. If you see 100k pseudo IDs but only 8k user IDs, that ratio should make you pause before you present any "unique users" number to a stakeholder.

Pitfall #9: ga_session_id Is Not Globally Unique

The Trap

Section titled “The Trap”

You need to do session-level analysis, so you pull `ga_session_id` from the event parameters and use it as your session key. You `GROUP BY` it, you `COUNT(DISTINCT)` it, you join on it. The numbers look reasonable enough, so you move on and build more logic on top. Weeks later, something weird surfaces — sessions with impossibly diverse page paths, or conversion rates that don’t add up. You dig in and realize that two completely different users, on different continents, sometimes share the same `ga_session_id`. Your “sessions” have been quietly merging unrelated visits.

Why It Happens

Section titled “Why It Happens”

The `ga_session_id` is a timestamp — specifically, the Unix timestamp (in seconds) of when the session started, generated on the client side. It’s scoped to a single `user_pseudo_id`, not globally unique across your entire dataset.

Think about it: if two users happen to start a session in the same second, they’ll get the same `ga_session_id`. On a busy site, this isn’t a rare coincidence — it’s a statistical certainty.

Google designed it this way because on the client, it only needs to be unique within one browser’s cookie jar. It was never meant to be a global session identifier. But when you pull all your data into BigQuery and treat it like one, that assumption breaks.

There’s another wrinkle. The `ga_session_id` can behave oddly around midnight boundaries and session timeout edges. If a user is active across a session timeout (30 minutes of inactivity by default), the new session gets a new timestamp — but I’ve seen cases where clock skew, timezone handling, or late-arriving events create unexpected overlaps. It’s rare, but it’s real.

The Fix

Section titled “The Fix”

Always use the composite key: `user_pseudo_id` + `ga_session_id`. Together, they form a genuinely unique session identifier.

Here's the pattern I use in every GA4 BigQuery project:

```
SELECT
  CONCAT(
    user_pseudo_id, '.',
    (SELECT value.int_value
     FROM UNNEST(event_params)
     WHERE key = 'ga_session_id')
  ) AS session_id,
  user_pseudo_id,
  event_name,
  event_timestamp
FROM
  `your_project.analytics_123456789.events_*`
WHERE
  _TABLE_SUFFIX = FORMAT_DATE(
    '%Y%m%d', CURRENT_DATE()
  );
```

That `CONCAT` gives you a single string you can safely `GROUP BY`, join on, and count. Make it a convention in your project — define it once in a view or a common CTE and reference it everywhere.

For extra robustness, especially on high-traffic properties, consider adding the `ga_session_number` (the sequential session count for that user) as a third component. This handles the rare edge case where a single user somehow gets two sessions with identical start timestamps:

```
CONCAT(
  user_pseudo_id, '.',
  (SELECT value.int_value
   FROM UNNEST(event_params)
   WHERE key = 'ga_session_id'),
  '.',
  (SELECT value.int_value
   FROM UNNEST(event_params)
   WHERE key = 'ga_session_number')
) AS session_id
```

Is the three-part key necessary for most teams? Probably not. But it costs nothing extra and closes the last remaining edge case. I'd rather over-engineer a primary key than debug phantom sessions six months from now.

How to Check If You Have This Problem

Section titled “How to Check If You Have This Problem”

This query finds `ga_session_id` values that appear under multiple `user_pseudo_id` values — direct evidence of collision:

```
WITH sessions AS (
  SELECT
    user_pseudo_id,
    (SELECT value.int_value
     FROM UNNEST(event_params)
     WHERE key = 'ga_session_id'
    ) AS ga_session_id
  FROM
    `your_project.analytics_123456789.events_*`
  WHERE
    _TABLE_SUFFIX BETWEEN
      FORMAT_DATE('%Y%m%d',
        DATE_SUB(CURRENT_DATE(), INTERVAL 7 DAY))
      AND FORMAT_DATE('%Y%m%d', CURRENT_DATE())
)

SELECT
  ga_session_id,
  COUNT(DISTINCT user_pseudo_id) AS user_count
FROM sessions
WHERE ga_session_id IS NOT NULL
GROUP BY ga_session_id
HAVING COUNT(DISTINCT user_pseudo_id) > 1
ORDER BY user_count DESC
LIMIT 20;
```

If this returns rows, you have collisions. On any reasonably busy property, it will. The `user_count` column shows you how many different users share each `ga_session_id` value. If you’ve been using `ga_session_id` alone as your session key, every one of those rows represents silently merged sessions in your analysis.

Pitfall #10: Consent Mode Creates Invisible Data Gaps

The Trap

Section titled “The Trap”

You’ve implemented Google Consent Mode v2 — the right thing to do for privacy compliance. Your data keeps flowing into BigQuery, your event counts look healthy, and everything seems fine. Then you try to do session analysis and the numbers are off. Or you notice your user count is lower than expected. Or — and this is the sneaky one — your conversion rate looks *higher* than it should, and you can’t figure out why.

What’s happening is that a chunk of your events are arriving without the identifiers you need. Consent mode is working exactly as designed, and it’s quietly making parts of your dataset unreliable for user-level and session-level analysis.

Why It Happens

Section titled “Why It Happens”

When a user doesn’t grant consent for analytics cookies, Google Consent Mode does something that’s easy to miss: it still sends events, but it strips out the identifying information.

In Advanced Consent Mode (which is what most implementations use), non-consented hits arrive in your BigQuery export as what I call “cookieless pings.” They have event names, timestamps, page locations — but the `user_pseudo_id` is either missing or replaced with a temporary value, and `ga_session_id` is absent. These events count toward your totals but can’t be stitched into sessions or attributed to users.

Here’s the thing — your event count stays high, which makes everything *feel* normal. But when you try to aggregate at the session or user level, you’re working with an incomplete picture. And the incompleteness isn’t random. Users who decline consent tend to be more privacy-conscious, often more technically sophisticated, and depending on your market, may represent a specific demographic segment.

So when teams filter to consented-only data — the natural instinct, since it’s the only data with usable identifiers — they’re not just shrinking the sample. They’re introducing a systematic bias toward users who click “Accept All.” In some European markets, I’ve

seen consent rates as low as 30-40%. Filtering to consented-only means you're basing your analysis on the minority of your traffic.

The Fix

Section titled "The Fix"

There's no magic query that makes this go away. The fix is a conscious strategy, and it starts with understanding your actual consent breakdown.

Step one: know your consent rate. Run the diagnostic below. This tells you what percentage of your data is fully consented, partially consented, and non-consented.

Step two: decide on a strategy based on that number:

- **High consent rate (>80%):** You can mostly filter to consented data and note the limitation. The bias exists but is manageable.
- **Medium consent rate (50-80%):** Use consented data for user/session analysis but use *all* events (including non-consented) for aggregate metrics like page views and event counts. Be explicit about which metrics use which dataset.
- **Low consent rate (<50%):** Your consented-only data is a minority sample. Aggregate event-level analysis on the full dataset. For user/session work, consider statistical adjustment or server-side collection as a complement.

The key principle: never mix consented and non-consented data in session or user analysis without knowing you're doing it. And never present consented-only analysis as "total" numbers without disclosing the gap.

For page-level or content analysis where you don't need user identity, use all events regardless of consent status:

```
-- Aggregate analysis: use all events
SELECT
  (SELECT value.string_value
   FROM UNNEST(event_params)
   WHERE key = 'page_location'
  ) AS page,
  COUNT(*) AS pageviews
FROM
  `your_project.analytics_123456789.events_*`
WHERE
  _TABLE_SUFFIX BETWEEN
    FORMAT_DATE('%Y%m%d',
      DATE_SUB(CURRENT_DATE(), INTERVAL 7 DAY))
    AND FORMAT_DATE('%Y%m%d', CURRENT_DATE())
  AND event_name = 'page_view'
GROUP BY page
ORDER BY pageviews DESC
LIMIT 50;
```

For session-level analysis, filter explicitly and document it:

```
-- Session analysis: consented traffic only
SELECT
  CONCAT(user_pseudo_id, '.',
    (SELECT value.int_value
     FROM UNNEST(event_params)
     WHERE key = 'ga_session_id')
  ) AS session_id,
  COUNT(*) AS events_in_session
FROM
  `your_project.analytics_123456789.events_*`
WHERE
  _TABLE_SUFFIX BETWEEN
    FORMAT_DATE('%Y%m%d',
      DATE_SUB(CURRENT_DATE(), INTERVAL 7 DAY))
    AND FORMAT_DATE('%Y%m%d', CURRENT_DATE())
  AND privacy_info.analytics_storage = 'Yes'
GROUP BY session_id
HAVING session_id IS NOT NULL;
```

How to Check If You Have This Problem

Section titled “How to Check If You Have This Problem”

Run this to see your consent breakdown right now:

```sql
SELECT
  privacy_info.analytics_storage,
  privacy_info.ads_storage,
  COUNT(*) AS event_count,
  COUNT(DISTINCT user_pseudo_id)
    AS unique_pseudo_ids,
  ROUND(
    COUNT(*)
    / SUM(COUNT(*)) OVER () * 100, 1
  ) AS pct_of_events
FROM
  `your_project.analytics_123456789.events_*`
WHERE
  _TABLE_SUFFIX BETWEEN
    FORMAT_DATE('%Y%m%d',
      DATE_SUB(CURRENT_DATE(), INTERVAL 7 DAY))
    AND FORMAT_DATE('%Y%m%d', CURRENT_DATE())
GROUP BY
  privacy_info.analytics_storage,
  privacy_info.ads_storage
ORDER BY event_count DESC;
```

Look at the rows where `analytics_storage` is `'No'` or `NULL`. That's your invisible data gap. If those rows represent more than 20% of your events, you need a deliberate strategy — not just a `WHERE` clause.

Also pay attention to the `unique_pseudo_ids` column for the non-consented rows. If that number is suspiciously low or shows a lot of `NULL` values, those events are truly anonymous — they can't be sessionized or attributed at all. That's the portion of your data that's essentially event-level only. Knowing the size of that segment is the first step to not being surprised by it.

Pitfall #11: Trying to Match BigQuery Numbers to GA4 Reports

The Trap

Section titled “The Trap”

You’ve done the work. You’ve written careful SQL, handled the nested schema, filtered by `_TABLE_SUFFIX`, and built a clean query for active users or sessions over a specific date range. Then you open the GA4 interface to validate your numbers — and they don’t match. Not even close. So you spend the next two weeks tweaking your query, adjusting timezone logic, trying different session definitions, convinced there’s a bug in your SQL. There isn’t. The numbers were never going to match.

Why It Happens

Section titled “Why It Happens”

Google’s own developer advocates have said publicly that BigQuery exports and GA4 UI reports are “not expected to be reconcilable.” That’s not a caveat buried in fine print — it’s the design.

Here’s why. The GA4 interface applies a stack of processing that simply doesn’t exist in the raw export:

- **Thresholding.** When Google Signals is enabled, the UI applies thresholding — suppressing rows where user counts are too low to prevent re-identification. This never happens in BigQuery. You get all the data; the UI hides some of it. That alone can create gaps in date-level or dimension-level comparisons.
- **Behavioral and conversion modeling.** The GA4 UI includes modeled data — estimated conversions and behavioral signals for users who declined analytics cookies or whose data was otherwise incomplete. BigQuery contains only observed, raw data. If Consent Mode is active and a chunk of your users denied consent, the UI fills in estimates where BigQuery shows nothing (see also Pitfall #10).
- **HyperLogLog++ for unique counts.** The GA4 UI uses HyperLogLog++ (HLL++), a probabilistic algorithm that estimates unique counts extremely fast but with a small margin of error (~1-2%). A `COUNT(DISTINCT)` in BigQuery is exact. On large datasets, this difference compounds — your BigQuery number will be precise, the UI’s will be an approximation. Neither is “wrong,” but they won’t agree.

- **Attribution black boxes.** GA4's reports use their own attribution logic — data-driven attribution, channel groupings, conversion counting windows — none of which are transparent or reproducible in SQL.
- **Traffic source scoping.** The top-level `traffic_source` field in BigQuery is user-scoped — it represents the *first* source/medium that acquired the user. This is not the same as session source/medium in the GA4 UI. If you're comparing "where did this session come from?" using `traffic_source`, you're looking at a fundamentally different thing. The newer `collected_traffic_source` field in the export schema helps bridge this gap — it captures the session-level attribution that's closer to what the UI reports.
- **Scoping differences.** The UI applies its own session scoping logic, engagement thresholds, and active user definitions. These don't map cleanly to anything in the raw event stream.
- **`gclid` misattribution.** Google Ads click identifiers get special treatment in the GA4 interface. Sessions with a `gclid` are attributed in ways that can differ from what the raw `traffic_source` fields in your export suggest.
- **Timezone discrepancies.** Your BigQuery export stores `event_timestamp` in UTC microseconds. GA4 reports use the timezone configured in your property settings. A session that straddles midnight in your local timezone might land on different dates in the UI vs. your query.

Each of these differences is small-ish on its own. Combined, they produce discrepancies of 10-30% on metrics like active users, sessions, and conversions. And there's no way to reverse-engineer the exact combination, because Google doesn't publish the full methodology.

I've seen teams burn weeks — sometimes longer — chasing exact reconciliation. Pulling in consultants, filing support tickets, rewriting queries from scratch. The answer was always the same: the numbers diverge by design.

The Fix

Section titled "The Fix"

Stop trying to reconcile. Seriously. Treat GA4 UI and BigQuery as two different instruments measuring overlapping but non-identical things.

- **GA4 UI** is your trend-spotting tool. Good for: "Are sessions trending up or down this month?" "Is this campaign performing roughly in line with last quarter?" It's pre-processed, modeled, and convenient. Use it for directional monitoring.

- **BigQuery** is your source for custom analysis. Good for: anything the UI can't do. Cross-channel attribution modeling, user-level journey analysis, connecting GA4 events with your own backend data, building audiences from raw behavioral signals.

The moment you accept this, you stop wasting time on reconciliation and start spending it on the analysis that only BigQuery can give you.

If stakeholders need a single "official" number, pick one source and commit to it. For most teams doing serious analytics, that source should be BigQuery — because you control the definitions, the logic is transparent, and you can audit every step. Just don't expect the GA4 UI to agree, and educate your stakeholders on why.

How to Check If You Have This Problem

Section titled "How to Check If You Have This Problem"

Run this query for a recent 7-day window:

```
SELECT
  FORMAT_DATE('%Y-%m-%d',
    DATE(TIMESTAMP_MICROS(event_timestamp),
         'America/New_York')  -- your GA4 property timezone
  ) AS event_date,
  COUNT(DISTINCT user_pseudo_id) AS bq_active_users,
  COUNT(DISTINCT
    CONCAT(user_pseudo_id, '.',
      (SELECT value.int_value
       FROM UNNEST(event_params)
       WHERE key = 'ga_session_id'))
  ) AS bq_sessions
FROM
  `your_project.analytics_123456789.events_*`
WHERE
  _TABLE_SUFFIX BETWEEN
    FORMAT_DATE('%Y%m%d',
      DATE_SUB(CURRENT_DATE(), INTERVAL 7 DAY))
    AND FORMAT_DATE('%Y%m%d', CURRENT_DATE())
GROUP BY event_date
ORDER BY event_date;
```

Now open the GA4 interface, navigate to Reports > Engagement > Overview (or the standard Users report), set the same date range, and compare day by day.

You'll see discrepancies. Probably 10-30%, sometimes more. The direction of the discrepancy may not even be consistent day to day — some days BigQuery will be higher, some days the UI will be.

That’s the point. Seeing this gap with your own data is what converts the intellectual understanding (“they’re not reconcilable”) into a lived experience that changes how you work. Once you’ve seen it, you stop chasing the match and start building things only BigQuery can do.

Pitfall #12: Rebuilding the GA4 UI in SQL

The Trap

Section titled “The Trap”

This is the single most common waste of time I see with GA4 BigQuery exports. A team gets access to the raw data, and the first thing they do is try to recreate the GA4 interface in SQL. They build queries for `session_start` counts. They reverse-engineer `session_engaged`. They write elaborate logic to calculate bounce rate from raw events. They spend weeks building a dashboard that shows… exactly what the GA4 interface already shows. Just slower, more expensive, and harder to maintain.

It feels productive. You’re writing SQL, you’re learning the schema, things are happening. But you’re not creating any new value. GA4 already has an interface that does this — and Google has an entire team maintaining it. You’re not going to out-build them on their own metrics.

Why It Happens

Section titled “Why It Happens”

There’s a strong gravitational pull toward the familiar. When you first open the raw export, the natural instinct is to reproduce what you already know — the reports you’ve been looking at in the GA4 interface. Session counts, page views, bounce rates. These feel like the “correct” starting point because they’re the metrics the organization already talks about.

There’s also a validation instinct at play. Teams want to prove the BigQuery data is “correct” by matching it to the UI. (Pitfall #11 explains why that match will never be exact, but the impulse persists.)

And sometimes it’s organizational: someone asks for a dashboard, and the dashboard spec is just “the GA4 reports, but in Looker.” Nobody stops to ask whether that’s a good use of the data team’s time.

The Fix

Section titled “The Fix”

Before you write a single query against the GA4 export, answer this question:

What business value do I need that GA4's interface cannot give me?

If you can't answer that clearly, you don't need BigQuery yet. And that's fine. GA4's Explore interface is genuinely capable for standard web analytics. Use it.

The things BigQuery is actually for — the reasons you went through the trouble of enabling the export — are the things the GA4 UI fundamentally cannot do:

- **User-level journey analysis** across sessions, devices, and time windows you define
- **Joining GA4 data with your own backend** — CRM records, subscription data, product usage events
- **Custom attribution modeling** where you control the logic, the window, and the definition of a conversion
- **Audience building from raw behavioral signals** that go beyond GA4's built-in segments
- **Feeding cleaned event data into ML pipelines** or other analytical tools

Build those things. Only those things. Everything else, let the GA4 interface handle.

This isn't about being lazy — it's about being deliberate. Every hour your team spends rebuilding bounce rate in SQL is an hour they're not spending on the analysis that only BigQuery can provide. That's the real cost.

How to Check If You Have This Problem

Section titled "How to Check If You Have This Problem"
This one doesn't need a SQL query. It needs a team conversation.

List every BigQuery query, scheduled model, or dashboard your team currently runs against GA4 data. For each one, ask a single question:

Does GA4's Explore interface already answer this?

If the answer is yes — and be honest — that query is a candidate for deletion. Not optimization, not refactoring. Deletion. Let the GA4 interface do what it was built to do, and redirect the engineering effort toward what it can't.

If you find that more than half your BigQuery GA4 work is recreating standard web analytics reports, that's a clear signal. Your team has been busy, but not on the right things. The export exists to unlock analysis the interface can't do — not to rebuild the interface at higher cost.

This connects directly to the Attribution Readiness Scorecard question #1: *Do you have defined use cases beyond GA4's interface?* If you don't, you haven't found the reason to use BigQuery yet. Find the reason first. Then build.

Pitfall #13: Over-Extracting Just in Case

The Trap

Section titled “The Trap”

You’re building your GA4 data model in BigQuery. You know the raw export is messy — nested arrays, hundreds of event parameters, cryptic key names. So you decide to be thorough. You unnest *everything*. Every event parameter, every user property, every item field. You extract all 200+ parameter keys into their own columns and pipe the whole thing into a staging table. Because what if someone needs `engagement_time_msec` six months from now? What if `entrances` turns out to be important? Better to have it and not need it than need it and not have it. Right?

Six months later, your staging table is enormous. Queries take minutes instead of seconds. Your BigQuery bill has a line item nobody wants to explain. And the worst part — nobody is using 90% of those columns. They never did.

Why It Happens

Section titled “Why It Happens”

It’s a fear-driven design choice, and I get it. The GA4 export has a genuinely intimidating number of parameters. When you run a quick count of distinct `event_params.key` values across your dataset, you’ll typically see 50-200+ unique keys. Some are standard GA4 parameters, some were set by your implementation, and some are mysterious leftovers from plugins or tag configurations nobody remembers setting up.

The instinct is to preserve everything because re-processing historical data is expensive. If you don’t extract a parameter now and realize you need it later, you’ll have to go back and scan the raw tables again. That’s a real cost, and it makes “extract everything” feel like the safe choice.

But “safe” has its own costs. Every additional column in your model means more bytes stored, more bytes scanned on every query, and more cognitive load for anyone trying to understand the table. A model with 200 columns is a model nobody can reason about. Analysts open it, see a wall of fields, and either pick the wrong ones or give up and ask someone for help.

The Fix

Section titled “The Fix”

Flip the sequence. Instead of extracting everything and hoping use cases emerge, define your use cases first, then identify exactly which parameters those use cases require.

Here’s the process:

1. **Start with the question.** What business questions does this model need to answer? Attribution? Funnel analysis? Churn prediction? Be specific.
2. **Map questions to fields.** For each question, identify the minimum set of event parameters needed. Attribution might need `source`, `medium`, `campaign`, `gclid`, `page_location`, and `ga_session_id`. That’s six parameters. Not sixty.
3. **Extract only those.** Build your model around the fields you actually need. Everything else stays in the raw `events_*` tables, accessible if a new use case genuinely emerges.

Look, if a new use case comes up in three months that needs a parameter you didn’t extract — you go back and add it. That’s a 20-minute change to your model. It’s not a disaster. The disaster is running a 200-column model for three months that nobody can understand and everyone pays for.

How to Check If You Have This Problem

Section titled “How to Check If You Have This Problem”

Run this to see how many distinct event parameter keys exist in your data:

```
SELECT
  params.key,
  COUNT(*) AS occurrences
FROM
  `your_project.analytics_123456789.events_*`,
  UNNEST(event_params) AS params
WHERE
  _TABLE_SUFFIX BETWEEN
    FORMAT_DATE('%Y%m%d',
      DATE_SUB(CURRENT_DATE(), INTERVAL 7 DAY))
    AND FORMAT_DATE('%Y%m%d', CURRENT_DATE())
GROUP BY params.key
ORDER BY occurrences DESC;
```

Now compare that list to the parameters your downstream queries actually reference. You can check this by searching your scheduled queries, dbt models, or Looker explores for the specific `event_params` keys they filter or extract.

If you have 150 parameter keys in the raw data but your actual queries only touch 12 of them, you know the ratio. Extracting those other 138 “just in case” is costing you money and clarity for zero return.

Pitfall #14: Not Reducing to Lean Event Tables

The Trap

Section titled “The Trap”

You’ve avoided pitfall #13. You’re not extracting everything. But you’re still running your analysis directly against the raw `events_*` tables — or against a lightly transformed version that’s basically the same schema with some columns renamed. Every query still scans nested arrays, processes hundreds of gigabytes, and takes thirty seconds to return. Your analysts grumble about query times. Your finance team grumbles about the bill. And nobody wants to iterate on an analysis that costs $2 every time you hit Run.

Why It Happens

Section titled “Why It Happens”

There’s a gap between “I know I shouldn’t extract everything” and “I’ve actually built the lean tables I need.” Pitfall #13 is about not pulling too much. This pitfall is about the positive design step that comes after — actually building purpose-built tables with only the fields your use cases require.

Many teams stop at the raw layer because building the clean layer feels like a big project. They want to “get it right” before committing to a table design. So they keep querying the raw data directly, telling themselves the clean model is coming “next sprint.” Months pass.

The Fix

Section titled “The Fix”

Once you’ve made your design decisions — identity strategy, touchpoint definition, use cases — you know exactly what you need. The answer is almost always surprisingly small.

A typical lean event table for attribution work might look like this:

```sql
CREATE OR REPLACE TABLE
  `your_project.analytics_123456789.lean_events`
PARTITION BY event_date
AS
SELECT
  PARSE_DATE('%Y%m%d', event_date) AS event_date,
  event_timestamp,
  event_name,
  user_pseudo_id,
  user_id,
  (SELECT value.int_value
   FROM UNNEST(event_params)
   WHERE key = 'ga_session_id')
    AS ga_session_id,
  (SELECT value.string_value
   FROM UNNEST(event_params)
   WHERE key = 'source')
    AS source,
  (SELECT value.string_value
   FROM UNNEST(event_params)
   WHERE key = 'medium')
    AS medium,
  (SELECT value.string_value
   FROM UNNEST(event_params)
   WHERE key = 'campaign')
    AS campaign,
  (SELECT value.string_value
   FROM UNNEST(event_params)
   WHERE key = 'page_location')
    AS page_location
FROM
  `your_project.analytics_123456789.events_*`
WHERE
  _TABLE_SUFFIX BETWEEN '20260101' AND '20260325';
```

That's it. Six extracted fields plus the standard columns. Everything is flat — no nested arrays, no UNNEST in downstream queries. Partitioned by date so queries scan only the days they need.

The difference is dramatic:

- **Cost.** Queries against this table will scan a fraction of the bytes. We're talking 10x-50x reduction depending on how many event parameters your raw data carries.
- **Speed.** Flat columns with partition pruning means sub-second response times for queries that used to take thirty seconds.
- **Clarity.** An analyst can open this table, see 10 columns with clear names, and immediately understand what's available. No digging through nested structs, no guessing which `value` subfield to use.

And here's a benefit that's becoming increasingly important: a lean table is what makes AI-assisted analysis possible. An LLM — whether it's powering a chatbot, a code assistant, or an autonomous agent — can reason about 6-10 clearly named columns. It cannot usefully reason about 200 nested event parameters. If you want to use agents or natural-language interfaces on your analytics data, lean tables aren't optional. They're a prerequisite.

Schedule this as a daily or weekly job (depending on your freshness requirements), and your analysts never touch the raw export again for routine work. The raw tables are still there for edge cases — but those should be rare.

How to Check If You Have This Problem

Section titled "How to Check If You Have This Problem"

Run the same analysis query twice — once against the raw table, once against a lean version — and compare the bytes processed and elapsed time.

First, check a typical query against raw:

```
-- Run this and note the bytes processed
SELECT
  PARSE_DATE('%Y%m%d', event_date) AS day,
  COUNT(DISTINCT user_pseudo_id) AS users,
  COUNT(DISTINCT
    CONCAT(user_pseudo_id, '.',
      (SELECT value.int_value
       FROM UNNEST(event_params)
       WHERE key = 'ga_session_id'))
  ) AS sessions
FROM
  `your_project.analytics_123456789.events_*`
WHERE
  _TABLE_SUFFIX BETWEEN '20260301' AND '20260325'
GROUP BY day
ORDER BY day;
```

Then run the equivalent against a lean table (after creating one):

```sql
SELECT
  event_date AS day,
  COUNT(DISTINCT user_pseudo_id) AS users,
  COUNT(DISTINCT
    CONCAT(user_pseudo_id, '.', ga_session_id)
  ) AS sessions
FROM
  `your_project.analytics_123456789.lean_events`
WHERE
  event_date BETWEEN '2026-03-01' AND '2026-03-25'
GROUP BY day
ORDER BY day;
```

Compare the bytes processed (shown in the query results panel) and the wall-clock time. If the lean table version processes 10x fewer bytes and returns in under a second, you've proven the case.

Also do a quick column audit on your current working tables. If any table your team queries regularly has more than 10-15 columns, ask yourself: which of these are actually used in downstream analysis? The rest is dead weight — adding cost, slowing queries, and making the schema harder to understand than it needs to be.

The Attribution Readiness Scorecard

You've just read through 14 pitfalls. Some probably felt familiar. Some might have prompted an uncomfortable "wait, are we doing that?" moment. Good. That discomfort is the beginning of building something better.

Before you jump into building attribution — multi-touch models, marketing mix analyses, all the sophisticated stuff — you need to honestly assess where you stand. Not where you wish you stood. Where you actually are, right now, with the data and infrastructure you have today.

These five questions emerged from working with dozens of teams on exactly this problem. They're the questions that separate teams who build attribution that works from teams who build attribution that looks impressive in a slide deck but nobody trusts.

Sit with each one. If you can, discuss them with your team. The conversation is as valuable as the answers.

---

1. Do you have defined use cases beyond GA4's interface?

Section titled "1. Do you have defined use cases beyond GA4's interface?"
This is question zero. If you can't articulate what you need from BigQuery that GA4's interface doesn't already provide, you don't have a reason to build a custom data model yet. And building without a reason is how you end up rebuilding the GA4 UI in SQL (Pitfall #12) and spending months on work that creates no new value.

A good answer sounds like: "We need to join GA4 behavioral data with our CRM to understand which marketing touchpoints drive accounts that actually retain past month three." That's specific. That's something the GA4 interface genuinely cannot do.

A bad answer sounds like: "We want to see our metrics in Looker instead of GA4." That's a visualization preference, not a use case. The GA4 interface already shows you those metrics.

---

2. Can you distinguish anonymous from account-level users?

Section titled "2. Can you distinguish anonymous from account-level users?"
Attribution is fundamentally about connecting a person's journey from first touch to conversion. If you can't tell anonymous visitors apart from identified accounts in your

data model, you're trying to trace journeys for entities you can't even define.

A good answer: "Yes. We have a clear identity model — anonymous traffic stays on `user_pseudo_id`, and once someone logs in or signs up, we map their pseudo IDs to a persistent account identifier. We know which pool each event belongs to."

A bad answer: "We use `user_pseudo_id` for everything." That means you're counting cookies, not people — and your attribution model is built on sand (Pitfall #8).

---

3. What's your identification rate, and do you have a plan to increase it?

Section titled "3. What's your identification rate, and do you have a plan to increase it?"
Your identification rate is the percentage of sessions or users you can tie to a known account. This number is your attribution ceiling. If only 15% of your sessions are ever identified, even a perfect attribution model can only explain 15% of the picture. The other 85% is a black box.

A good answer: "Our identification rate is 35%, and we're running experiments with email URL parameters and post-login session stitching to push it toward 50% by Q3."

A bad answer: "I don't know." Or: "We haven't measured it." If you haven't measured it, you don't know how much of your data you're blind to — and any attribution model you build is working with an unknown fraction of reality. Run the identity coverage query from Pitfall #8 before you go any further.

---

4. How do you define a touchpoint, and how many does your average account have?

Section titled "4. How do you define a touchpoint, and how many does your average account have?"
A touchpoint is the unit of your attribution model. If you haven't defined what counts as one, you can't build attribution. And if you have defined it, the average count per account tells you how complex your model needs to be.

A good answer: "A touchpoint is a session with a marketing attribution source. Our average identified account has 4.2 touchpoints before conversion. We also capture post-purchase survey responses as virtual touchpoints."

A bad answer: "Every page view is a touchpoint." That's not a touchpoint definition — that's raw event data. You'll drown in noise. Or: "Our average account has 1.3 touchpoints." If most accounts have only one or two touchpoints, you don't need multi-

touch attribution. Single-touch is simpler, cheaper, and tells you everything the data can support. Save the complexity for when the data warrants it.

---

5. What percentage of your conversions can you actually attribute end-to-end?

Section titled “5. What percentage of your conversions can you actually attribute end-to-end?”

This is the punchline. After identity resolution, after touchpoint design, after connecting everything together — what fraction of your actual conversions have a complete, traceable journey from first touch to conversion?

A good answer: “62% of our conversions have at least one attributable touchpoint. 40% have three or more, which is where our multi-touch model adds value. We know the other 38% are a gap, and we’re working on closing it through better identification.”

A bad answer: “We assume it’s high because our attribution model produces numbers.” That’s not coverage — that’s faith. If you haven’t measured the gap between total conversions and attributable conversions, your model might be explaining 30% of the picture while presenting it as 100%. That’s worse than no model at all, because it creates false confidence.

What to Do with Your Answers

Section titled “What to Do with Your Answers”

If you answered all five confidently, with real numbers, you’re ready to build attribution. Genuinely ready — not “we enabled the BigQuery export” ready, but “we understand our data, our gaps, and our limitations” ready. That’s a strong position.

If some answers made you uncomfortable, that’s the point. Each gap maps to a specific piece of work — an identity model to build, a coverage metric to measure, a use case to define. Those are your next steps. They’re not glamorous, but they’re the foundation that makes everything else work.

The ebook you just read covered the pitfalls. The scorecard shows you where you are. Part 2 of this training series covers what comes next: building the actual attribution model on top of this foundation.

About Propel & What's Next

About Propel

Section titled “About Propel”

Propel helps B2B companies build data infrastructure for growth analytics — from GA4 data modeling to full attribution systems. We work with data teams who are past the “we have data” stage and into the “we need to make this data actually useful for business decisions” stage.

This ebook came out of patterns we see repeatedly across client engagements. The pitfalls are real, the SQL is tested, and the scorecard questions are the same ones we work through with every new team we partner with.

I’m Timo Dechau, and this is what I spend my days on — helping teams build the data foundations that make attribution, forecasting, and growth analysis actually trustworthy. Not dashboards that look good. Systems that work.

What’s Next

Section titled “What’s Next”

Part 2: Attribution Modeling

This ebook covered the foundation — the pitfalls, the design decisions, the honest self-assessment. Part 2 of the training series is where we build the actual attribution model on top of it. Multi-touch modeling, channel contribution analysis, and the practical SQL to make it work.

If the scorecard questions resonated, Part 2 is where you’ll see how those answers turn into an attribution system you can actually trust.

Need help with your team’s GA4 data model?

If you’re reading this and thinking “we should really sort this out properly” — that’s literally what we do. We work with data teams to build clean GA4 data models, identity resolution layers, and attribution systems. No twelve-month consulting engagements. Focused, practical work that gets your foundation right.

If that’s useful, let’s talk.